Proceedings of the Society for Computation in Linguistics

Volume 6 Article 7

6-1-2023

CANDS: A Computational Implementation of Collins and Stabler (2016)

Satoru Ozaki
University of Massachusetts Amherst, sozaki@umass.edu

Yohei Oseki
The University of Tokyo, oseki@g.ecc.u-tokyo.ac.jp

Follow this and additional works at: https://scholarworks.umass.edu/scil

 Part of the Computational Linguistics Commons, and the Syntax Commons

Recommended Citation

Ozaki, Satoru and Oseki, Yohei (2023) "CANDS: A Computational Implementation of Collins and Stabler (2016)," *Proceedings of the Society for Computation in Linguistics*: Vol. 6, Article 7.
DOI: https://doi.org/10.7275/908h-3s86
Available at: https://scholarworks.umass.edu/scil/vol6/iss1/7

This Paper is brought to you for free and open access by ScholarWorks@UMass Amherst. It has been accepted for inclusion in Proceedings of the Society for Computation in Linguistics by an authorized editor of ScholarWorks@UMass Amherst. For more information, please contact scholarworks@library.umass.edu.

CANDS: A Computational Implementation of Collins and Stabler (2016)

Satoru Ozaki
University of Massachusetts Amherst
sozaki@umass.edu

Yohei Oseki
University of Tokyo
oseki@g.ecc.u-tokyo.ac.jp

Abstract

Syntacticians must keep track of the empirical coverages and the inner workings of syntactic theories, a task especially demanding for minimalist syntacticians to perform manually and mentally. We believe that the computational implementation of syntactic theories is desirable in that it not only (a) facilitates the evaluation of their empirical coverages, but also (b) forces syntacticians to specify their inner workings. In this paper, we present CANDS, a computational implementation of **C**ollins **AND** **S**tabler (2016) in the programming language Rust. Specifically, CANDS consists of one main library, `cands`, as well as two wrapper programs for `cands`, `derivck` and `derivexp`. The main library, `cands`, implements key definitions of fundamental concepts in minimalist syntax from Collins and Stabler (2016), which can be employed to evaluate and extend specific syntactic theories. The wrapper programs, `derivck` and `derivexp`, allow syntacticians to check and explore syntactic derivations through an accessible interface.[1]

1 Introduction

Syntax typically involves developing a new theory or revising an existing theory in order to explain certain data. A syntactician needs to be able to compare the theories in terms of their empirical coverage and understand all the details of these theories. These are challenging prerequisites to attain for minimalist syntacticians (Chomsky, 1995). This is partly due to the lack of consensus on the exact mechanism of minimalist syntactic theory, despite many efforts to formalize it (e.g., Veenstra 1998; Kracht 1999, 2001, 2008; Frampton 2004; Collins and Stabler 2016), and partly due to the constant source of subtle revisions to this theory.

We believe that the computational implementation of syntactic theories would help minimalist syntacticians understand their empirical coverages and inner workings. This idea has been explored in the LFG and HPSG literature with their rich histories of grammar engineering (e.g., Bierwisch 1963; Zwicky et al. 1965; Müller 1999; Butt 1999; Bender et al. 2002, 2008, 2010; Fokkens 2014; Müller 2015; Zamaraeva 2021; Zamaraeva et al. 2022). In comparison, there is less effort on the computational implementation of syntactic theories in the minimalist literature, with some exceptions (e.g., Fong and Ginsburg, 2019). In this paper, we present CANDS (pronounced /kændz/), a computational implementation of **C**ollins **AND** **S**tabler (2016) (henceforth C&S) in the programming language Rust. The main library, `cands`, implements key definitions of fundamental concepts in minimalist syntax from Collins and Stabler (2016), which itself is a formalization of minimalist syntax. We hope that `cands` can be employed to evaluate and extend specific syntactic theories.

In addition, to make `cands` accessible to minimalist syntacticians who are not familiar with Rust, we also provide two wrapper programs for `cands` which allow syntacticians to check and explore syntactic derivations through an accessible interface: the derivation checker `derivck`, and the derivation explorer `derivexp`.

This paper is organized as follows. In Section 2, we review key definitions of fundamental concepts in minimalist syntax from C&S. In Section 3, we introduce the main library, `cands`, as well as two wrapper programs, `derivck` and `derivexp`, illustrating their usage with example codes and screenshots. In Section 4, we demonstrate how `cands` can be employed to evaluate syntactic theories with two particular formulations of the Subject Condition. In Section 5, we show how `cands` can be used to extend syntactic theories with two particular implementations of the syntactic operation Agree. We discuss future work in Section 6 and conclude the paper in Section 7.

[1] Our software is available at https://github.com/osekilab/CANDS.

Proceedings of the Society for Computation in Linguistics (SCiL) 2023, pages 47-68.
Amherst, Massachusetts, June 15-17, 2023

2 Collins and Stabler (2016)

Collins and Stabler (2016) provide a precise formulation of minimalist syntax. In this section, we review some key definitions in their work.

Universal Grammar (UG) is a 6-tuple $\langle \text{PHON-F}, \text{SYN-F}, \text{SEM-F}, \text{Select}, \text{Merge}, \text{Transfer} \rangle$, where the first three elements specify the universal sets of phonological, syntactic and semantic features respectively, and the last three elements are syntactic operations.

An *I-language* is as a 2-tuple $\langle \text{Lex}, \text{UG} \rangle$ where Lex is a lexicon, i.e., a finite set of lexical items, and UG is some Universal Grammar.

A *lexical item* (LI) is a 3-tuple $\langle \text{SEM}, \text{SYN}, \text{PHON} \rangle$, where $\text{SEM} \subseteq \text{SEM-F}$, $\text{SYN} \subseteq \text{SYN-F}$ and $\text{PHON} \in \text{PHON-F}^*$. [2]

A *lexical item token* (LIT) is a 2-tuple $\langle \text{LI}, k \rangle$, where LI is a LI and k an *index*. This index is used to distinguish between multiple occurrences of the same LI related by movement.

Syntactic objects (SO) are inductively defined. A SO is one of three things: (a) a LIT, (b) the result of the syntactic operation $\text{Cyclic-Transfer}(\text{SO})$ for some syntactic object SO, or (c) a set of SOs.

A *lexical array* (LA) is a set of LITs, and a *workspace* W is a set of SOs. A *stage* is a 2-tuple $\langle \text{LA}, W \rangle$ of lexical array LA and workspace W.

The syntactic operations Select, Merge and Transfer are defined as functions. For example, for some stage $S = \langle \text{LA}, W \rangle$ and LIT $A \in \text{LA}$,

$$\text{Select}(A, S) = \langle \text{LA} \setminus \{A\}, W \cup \{A\} \rangle.$$

Cyclic-Transfer, which was used in the above definition of SOs, is a special unary case of Transfer, which is a binary operation.

The central definition in C&S is that of a derivation. A sequence of stages $S_1, \cdots, S_n$ with each $S_i = \langle \text{LA}_i, W_i \rangle$ is a *derivation* from lexicon L if (a) all LIs from the initial lexical array LA_1 come from L, (b) the initial workspace W_1 is empty, and (c) each subsequent stage S_{i+1} is derived from the previous stage S_i by an appropriate application of some syntactic operation. The conditions involved in (c) limit the generative capacity of the theory. For example, the conditions on Merge enforce that, if S_{i+1} is derived from S_i by $\text{Merge}(A, B)$, then $A \in W_i$, and either A contains B or $B \in W_i$. The first disjunct "A contains B" allows internal Merge, and the second disjunct "$B \in W_i$" allows external Merge. Certain patterns of Merge, such as sideward Merge, are disallowed in this formulation.

[2] PHON-F^* is the set of (potentially empty) sequences whose elements come from PHON-F, i.e., $\bigcup_{k=0}^{\infty} \text{PHON-F}^k$.

3 CANDS

CANDS consists of the main library, `cands`, and two wrapper programs for `cands`, `derivck` and `derivexp`. They are all developed in the programming language Rust.

3.1 `cands`

`cands` is a library that implements and exposes most concepts defined in C&S. We provide a full list of implemented definitions in Appendix A.

Figure 1 shows the Rust code that uses `cands` to create a SO. This SO is a LIT, with index 37 and a LI that consists of the semantic features {[M]}, the syntactic features {[D]}, and the phonological features ⟨[Mary]⟩. `SyntacticObject` is an enum type defined in `cands`, which comes in three variants: LITs, sets and results of Cyclic-Transfer. Here, we use `SyntacticObject::LexicalItemToken` to construct a LIT variant. `cands` also defines the struct types `LexicalItemToken`, `LexicalItem` and `Feature`, each of which is associated with a `new` function that constructs an object of each type. `Set` and `Vec` are container types defined in the Rust standard library, and their associated `from` functions create sets and vectors.[3]

```
SyntacticObject::LexicalItemToken(
  LexicalItemToken::new(
    LexicalItem::new(
      Set::from([Feature::new("M")]),
      Set::from([Feature::new("D")]),
      Vec::from([Feature::new("Mary")])
    ), 37
  )
)
```

Figure 1: Code to create a SO.

`cands` defines many macros, which help reduce boilerplate code. For example, `SyntacticObject::LexicalItemToken(...)` can be reduced to a much shorter macro invocation `so!(...)`. Similarly, LIs and LITs can be created with the macros `li!` and `lit!`

[3] To be precise, the Rust standard library does not define a set type called `Set`; rather, it defines two concrete implementations of a set type called `HashSet` and `BTreeSet`. `Set` is a type alias defined in `cands` that refers to `BTreeSet`.

respectively. Sets and vectors of features can be created with `fset!` and `fvec!`. The same code can be re-written more concisely as in Figure 2.

```
1 so!(lit!(li!(fset!( "M" );
2               fset!( "D" );
3               fvec![ "Mary" ]), 37))
```

Figure 2: Shorter code to create a SO.

An important feature of `cands` is the function `is_derivation`. This function implements the definition of derivations from C&S. It takes two arguments: `il`, of type `ILanguage`, which represents an I-language, and `stages`, of type `Vec<Stage>`, which represents a sequence of stages. `is_derivation(il, stages)` returns true iff `stages` is a derivation from `il` according to the definition in C&S.

We see two major usages of `cands`. First, it can be used to explore predictions from C&S. For example, one can check if a given sequence of stages is a valid derivation. Second, it can be extended to implement other notions and theories. C&S lacks formalization for many concepts that are popular in minimalist syntax, e.g., Agree, head movement and covert movement (Collins and Stabler, 2016). The predictions and empirical coverage of extensions to `cands` can be evaluated in a similar manner to the original `cands`.

3.2 Two wrapper programs for `cands`

Using `cands` requires programming in Rust, a relatively unfamiliar programming language among syntacticians. In order to make `cands` more accessible to the general audience, we provide two wrapper programs for `cands`. They are (a) `derivck`, a derivation checker that runs in the terminal, and (b) `derivexp`, an interactive derivation explorer that displays a GUI.

Figure 3 shows how the wrappers can be executed in a shell. Both programs require the user to provide an I-language IL and a sequence of stages S, both specified in JSON. These files are passed to the programs via command line arguments.

```
1 > derivck -i IL.json -d S.json
2 > derivexp -i IL.json -d S.json
```

Figure 3: Typical shell commands used to run `derivck` (line 1) and `derivexp` (line 2). The files specifying the I-language and the sequence of stages are passed via command line arguments.

`derivck` will output whether S is a derivation from IL. If not, `derivck` will display the offending stage(s) of S and a log that describes how it determined the stage(s) to be invalid. The log verbosity can be set with an environmental variable.

`derivexp` will first verify that S is a valid derivation. Then, it provides an interface that visualizes S and allows the user to apply various syntactic operations to the objects that comprise S to further advance the derivation. Figures 4a and 4b show screenshots from a `derivexp` session before and after the user has applied Merge to a pair of SOs.

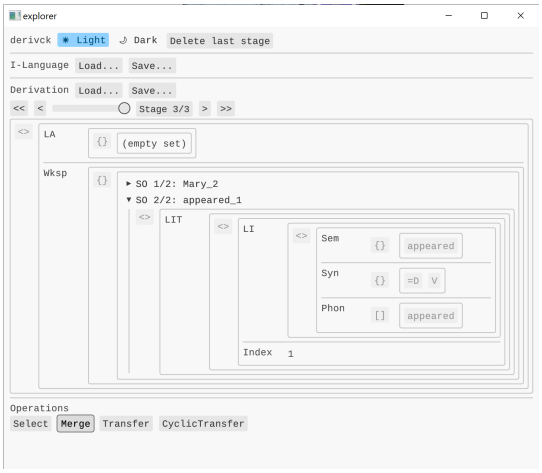
Both `derivck` and `derivexp` expect the JSON files for the I-language and the sequence of stages to be in a specific format that transparently reflects the Rust types for these two concepts, which are `ILanguage` and `Vec<Stage>`. This format is imposed by `serde`, a popular Rust data (de)serialization framework, which is used in `cands` to support human-readable JSON (de)serialization for its data structures. Even though we believe this format should be straightforward for users to follow, larger I-languages and sequences of stages in real-life use cases can be unwieldy to specify manually in JSON. In the near future, we plan to develop tools that would simplify the creation of these JSON files, such as a visual interface for constructing I-languages and sequences of stages and exporting them to JSON. For now, we provide sample JSON files in the Git repository for CANDS that can be used to construct a derivation for the simple sentence *Mary appeared*, as illustrated in Figure 4.[4]

We hope that `derivck` and `derivexp` will be useful for syntacticians working with the C&S system. If one already has a derivation in mind, they can check the derivation with `derivck`. Otherwise, one can use `derivexp` to explore the possible derivations generated by the C&S system. The two programs should facilitate working with grammatical and ungrammatical examples respectively.

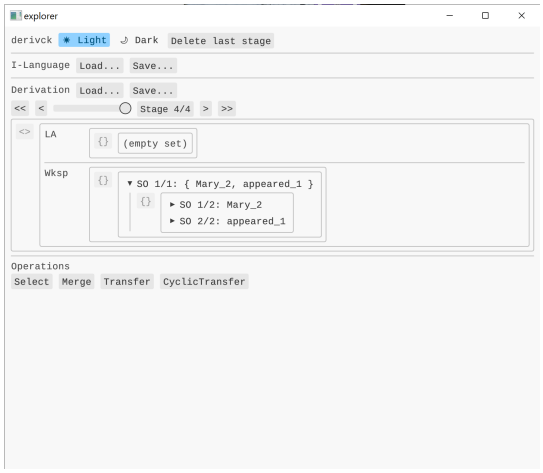
4 Evaluating theories with `cands`

An important and challenging task for syntacticians is to keep track of the empirical coverage of the syntactic theory at hand as one proposes changes to the theory. Often, one proposes a revision to the theory in order to make a correct prediction for one

[4]We thank one reviewer for pointing out the necessity to address how easily these JSON files can be created.



(a) `derivexp` is showing a stage S_3, whose workspace W_3 contains two roots: *Mary*$_2$ and *appeared*$_1$. We then apply Merge(*appeared*$_1$, *Mary*$_2$) to derive the next stage.



(b) We advance to the next stage S_4, whose workspace W_4 contains just one root, which is the result of Merge(*appeared*$_1$, *Mary*$_2$). `derivexp` is showing S_4.

Figure 4: Screenshots from a `derivexp` session.

sentence, only to realize later that another sentence correctly predicted by the old theory receives an incorrect prediction under the new theory.

Computational implementation of syntactic theories facilitates the process of examining their predictions and evaluating their empirical coverage. Using the function `is_derivation` defined in `cands`, it is easy to check if some derivation of interest can be generated by the C&S system. Even if one modifies `cands` in order to implement their revisions of C&S, predictions can be studied in the same way as long as the `is_derivation` function is preserved. Multiple revisions to C&S can be evaluated in terms of their empirical coverage by testing the corresponding modified versions of `cands` on a common set of derivations.

In this section, we illustrate this evaluation process with a simple example as a proof of concept. We consider the sentences in (1) and provide a derivation for each sentence. The original C&S system generates all three derivations, which is not ideal – we expect a good theory to only generate the derivations for the grammatical sentences. We will provide two attempts at positing a new constraint and incorporating it into C&S to correct the predictions. We will implement the new constraints as extensions of `cands`, and test these extensions on our derivations of interest. We will see that both attempts are inadequate in that each constraint fixes the prediction for one sentence while breaking the prediction for another. Our examples and analyses are inspired by classic literature on PP extraposition (Akmajian, 1975; Guéron, 1980; Wexler and Culicover, 1980).[5] For space reasons, we will only define the constraints and discuss their predictions conceptually in the main paper. We provide pseudocode for the implementations of these constraints in Appendix B, and the implementations themselves in the Git repository on the branches `theory1` and `theory2`.

(1) a. * A story bothered me about Mary.
b. A story appeared about Mary.
c. * I know who a story appeared about.

In (1a), PP extraposition occurs from the subject of a transitive verb. In (1b), the extraposition occurs from the subject of an unaccusative verb. In (1c), the same extraposition found in (1b) occurs, as well as *wh*-movement to embedded [Spec; CP].

The original C&S system allows for all three sentences to be derived, with the derivations sketched in (2), (3) and (4),[6] and fully detailed in Appendix C. To accommodate rightward extraposition in the LCA-like linearization algorithm employed in C&S, we use two covert heads X and Y as well as remnant movement. For example, in (2), X first merges with TP. The extraposed PP then moves to [Spec; XP]. Y then merges with XP, and the remnant TP moves to [Spec; YP]. Y contains the

[5] We thank Kyle Johnson for introducing us to the debate on PP extraposition when we were in search of syntactic phenomena to illustrate the usage of `cands` with.

[6] Non-final occurrences of SOs are struck out. Although the SOs in these derivations are actually sets, which should be denoted with comma-separated lists of elements enclosed in braces, we use the labelled bracket notation here to save space.

syntactic feature [T], which allows C to merge with YP as it would merge with a TP. In (2), the extraposition occurs from TP, while in (3) and (4) the extraposition occurs from VP.

Note on notations: we write $A \in^+ B$ for "A is contained in B", and $A \in^* B$ for "A is equal to or contained in B".

4.1 Theory 1: derivational constraint

Consider the pair (2) and (3). They differ in their grammaticality as well as the source of extraposition: the subject in the former derivation and the VP in the latter. We can predict these derivations correctly if we use a derivational flavor of the Subject Condition, i.e., a constraint that bans movement out of [Spec; TP]. Let us write $\mathrm{occ}_R(X)$ for the set of all occurrences of X in R. Then, we can add the condition (5) to the derive-by-Merge condition.

(5) **Derivational Subject Condition (DSC)**
For (internal) $\mathrm{Merge}(A, B)$ where A is the head and $B \in^+ A$, then $|\mathrm{occ}_A(B)| > \sum_{X \in \mathrm{Sbjs}_A(B)} |\mathrm{occ}_A(X)|$, where $\mathrm{Sbjs}_A(B)$ is the set of all [Spec; TP]s in A that contain B.

Consider internal $\mathrm{Merge}(A, B)$ where $B \in^+ A$. DSC holds iff there exists some occurrence B_P of B in A that is not equal to or contained in any occurrence of some [Spec; TP] in A. Thus, DSC holds iff this instance of Merge could be interpreted as movement from a non-subject position.

We call the C&S system extended by DSC "Theory 1." We implement and test Theory 1 against our derivations. The results show that only (2) is ungrammatical, so Theory 1 makes an incorrect prediction for (4). The PP extraposition in (2) violates DSC because all occurrences of the PP prior to this extraposition are contained under some occurrence of DP, which is at [Spec; TP]. The extrapositions in (3) and (4) do not violate DSC because the extraposition occurs before TP is even built. The subsequent *wh*-movement in (4) does not violate DSC either, due to the occurrence of *who* contained in the extraposed PP at [Spec; XP].

4.2 Theory 2: representational constraint

Consider the pair (2) and (4). They are both ungrammatical, and in both derivations there is a SO that has one occurrence inside and another occurrence outside of the subject, namely the PP *about Mary/who*. This suggests that perhaps the Subject Condition should be representational after all; any SO that has an occurrence inside some [Spec; TP] cannot have an occurrence outside that [Spec; TP]. This condition, formally stated as (6), is enforced at every stage of the derivation, applying to every workspace W_i.

Derivations	Truth	Theory 1	Theory 2
(2), for (1a)	*	*	*
(3), for (1b)	✓	✓	*
(4), for (1c)	*	✓	*

Table 1: Derivations, grammaticalities and predictions.

(6) **Representational Subject Condition (RSC)**
For any root $R \in W_i$ and any SOs $X, S \in^* R$ such that $X \in^* S$ and S is [Spec; TP], $|\mathrm{occ}_R(X)| = |\mathrm{occ}_R(S)|$.

If $X \in^* S \in^* R$, then every occurrence of S in R is either equal to or contains some occurrence of X in R (Theorem 1 from C&S). Thus $X \in^* S \in^* R$ implies $|\mathrm{occ}_R(X)| \geq |\mathrm{occ}_R(S)|$. If $|\mathrm{occ}_R(X)| > |\mathrm{occ}_R(S)|$, it must be the case that some occurrence of X is not equal to or contained in any occurrence of S. This is exactly the situation that RSC bans.

Let us call the C&S system extended by RSC "Theory 2". We implement and test Theory 2 against our derivations. Although Theory 2 correctly rules out (2) and (4), it incorrectly rules out (3) as well. This is because at the final stage in all three derivations, the PP *about Mary/who* has four occurrences, while the [Spec; TP] *a story about Mary/who*, which contains the PP, has three occurrences.

Table 1 summarizes the derivations, their desired grammaticalities and the grammaticalities predicted by our theories.

5 Extending theories with `cands`

In the literature, minimalist syntactic theories are usually described in text, with various degrees of formality. As such, it can be difficult to communicate the precise details of the theories to the reader. The benefit of implementing theories in code is that one is forced to consider and specify such details, because otherwise one would end up with an incomplete implementation.

Since C&S is a formalization of a bare-bones Minimallist syntactic theory, we expect that `cands` will provide a good starting point for min-

(2) a. Build TP.
[TP [DP a story [PP about Mary]] T bothered me]
b. Extrapose PP.
[YP [TP [DP a story ~~[PP about Mary]~~] T bothered me] Y [XP [PP about Mary] X ~~TP~~]]

(3) a. Build VP.
[VP appeared [DP a story [PP about Mary]]]
b. Extrapose PP.
[YP [VP appeared [DP a story ~~[PP about Mary]~~]] Y [XP [PP about Mary] X ~~VP~~]]
c. Build TP; move DP.
[TP [DP a story ~~PP~~] T [YP [VP appeared ~~DP~~] Y [XP [PP about Mary] X ~~VP~~]]]

(4) a. Same with (3) up to (3c), except we have *who* instead of *Mary*.
[TP [DP a story ~~PP~~] T [YP [VP appeared ~~DP~~] Y [XP [PP about who] X ~~VP~~]]]
b. Build CP; move *who*.
[CP who Q [TP [DP a story ~~PP~~] T [YP [VP appeared ~~DP~~] Y [XP [PP about ~~who~~] X ~~VP~~]]]]

imalist syntacticians to implement their own proposals and theories on top of it. To illustrate this, we implement two proposals for Agree, a syntactic operation commonly assumed by minimalist syntacticians but is undefined in C&S. Specifically, we implement two proposals, described respectively in Chomsky 2001 and Collins 2017. Our implementations can be found on the Git repository on branches `agree-chomsky-2001` and `agree-collins-2017`. We recognize that there are many other proposals for Agree, such as Pesetsky and Torrego 2007, Béjar and Rezac 2009, Zeijlstra 2012, Preminger 2014 and Deal 2015.

5.1 Agree à la Chomsky (2001)

First, we formalize and implement Chomsky's (2001) proposal for Agree.

Our system distinguishes two kinds of syntactic features: *normal syntactic features*, which are just like semantic and phonological features; and *valuable syntactic features*, which are associated with interpretability and a potential value.

(7) A syntactic feature is either normal or valuable.

a. A *normal syntactic feature* is some $F \in \text{SYN-F}$.

b. A *valuable syntactic feature* is some $F = \langle i, f, v \rangle$ where $i \in \{\text{i}, \text{u}\}$ is its *interpretability*, $f \in \text{SYN-F}$, and either $v = \_$ (*unvalued*) or $v = v'$ for some value v' (*valued*). F is usually denoted $[if{:}v]$ (e.g. [uCase:_], [iPerson:3]).

Agree is a function that takes two LITs, which we call the *probe* and the *goal*. The probe is valued with the features from the goal, and if the probe is not defective, the goal is valued with the features from the probe. Agree returns the new probe and the new goal.

(8) For lexical item tokens

$$P = \langle \langle \text{SEM}_P, \text{SYN}_P, \text{PHON}_P \rangle, k_P \rangle,$$
$$G = \langle \langle \text{SEM}_G, \text{SYN}_G, \text{PHON}_G \rangle, k_G \rangle,$$

$\text{Agree}(P, G) = \langle P', G' \rangle$ where

$$P' = \langle \langle \text{SEM}_P, \text{SYN}_{P'}, \text{PHON}_P \rangle, k_P \rangle,$$
$$G' = \langle \langle \text{SEM}_G, \text{SYN}_{G'}, \text{PHON}_G \rangle, k_G \rangle,$$

where

$$\text{SYN}_{P'} = \{\text{Value}(F, \text{SYN}_G) \mid F \in \text{SYN}_P\},$$
$$\text{SYN}_{G'} = \text{SYN}_G \text{ if } P \text{ is defective, otherwise}$$
$$= \{\text{Value}(F, \text{SYN}_P) \mid F \in \text{SYN}_G\}.$$

(9) For a syntactic feature F and a set of syntactic features SYN, $\text{Value}(F, \text{SYN}) =$

a. F, if F is normal or valued.

b. $\langle i, f, v' \rangle$, if $F = \langle i, f, v \rangle$ with $v = \_$, and there is $F' = \langle i', f', v' \rangle \in \text{SYN}$.

We modify Clause (iii) of the C&S definition of derivations by adding the derive-by-Agree condition, which checks if a workspace W_{i+1} can be derived from the previous workspace W_i by applying Agree to an appropriate probe-goal pair.

(10) (derive-by-Agree) Consider the ith workspace W_i. Fix some $R \in W_i$ and some active, matching pair of lexical item tokens P, G such that

a. P c-commands G, and

b. for any lexical item token $H \in^* R$ such that H matches P and P c-commands H, either $G = H$ or G c-commands H.

Let $\langle P', G' \rangle = \text{Agree}(P, G)$, and let $X = R$, except all occurrences of P and G are respectively replaced with P' and G'. Then the next workspace W_{i+1} is *derived-by-Agree* from W_i if $W_{i+1} = (W \setminus \{R\}) \cup \{R'\}$, where either

a. $R' = X$ and P doesn't contain the EPP-feature, or

b. $R' = \text{Merge}(X, Y)$ and P contains the EPP-feature, with some Y that satisfies $G \in^* Y \in^* X$ determined by pied-piping.

Derivation by Agree necessarily changes SOs in place, thereby violating the No-Tampering Condition (NTC; Chomsky 2007). As a result, upon finding an appropriate probe-goal pair, our implementation of derive-by-Agree visits the entire structure of R in order to construct X from R by replacing the old probe and goals with new ones.

During the construction of X, it is necessary to replace all occurrences of the goal G with the new goal G', rather than just replacing the highest occurrence. This is common practice in a multidominance-based theory like C&S. Otherwise, the highest occurrence of the post-Agree goal will no longer be considered as the same SO as the remaining occurrences, which has consequences in linearization.

We illustrate our implementation with (11), a derivation for the sentence *The man falls*.[7] The full derivation is in Appendix D.

5.2 Agree à la Collins (2017)

Next, we formalize and implement Collins' (2017) proposal for Agree. This proposal differs from Chomsky 2001 in two important ways: (a) Agree is not its own syntactic operation, but rather a special case of Merge, and (b) derivation by "Agree" complies with the NTC and does not modify SOs in-place; rather, features are Merged to feature-checking positions.

[7] π is Person, # is Number and C is Case.

As with our implementation of Chomsky 2001, we split syntactic features into *normal syntactic features* and *valuable syntactic features*. In this implementation, however, the value of valuable syntactic feature is required. Unlike Chomsky's feature valuation system, Collins's feature checking system does not allow features to be unvalued.

(13) A syntactic feature is either normal or valuable.

a. A *normal syntactic feature* is some $F \in \text{SYN-F}$.

b. A *valuable syntactic feature* is some $F = \langle i, f, v \rangle$ where $i \in \{\text{i}, \text{u}\}$ is its interpretability, $f \in \text{SYN-F}$, and v is some value.

We redefine SOs so that they can be created by Merging a SO and a syntactic feature.[8]

(14) X is a syntactic object iff

a. X is a lexical item token, or

b. $X = \text{Cyclic-Transfer}(\text{SO})$ for some syntactic object SO, or

c. X is a set of syntactic objects, or

d. $X = \{\text{SO}, F\}$ for some syntactic object SO and syntactic feature F.

As we redefine SOs, we must also change many definitions that depend on SOs. A crucial example is Triggers; just as some Triggers function T is able to check a feature off a SO if it is Merged with another approriate SO, T should able to check an uninterpretable feature off a SO if it is Merged with an appropriate syntactic feature. We change Clause (ii) in the definition of Triggers that handles SOs of the type $\{\text{SO}, F\}$:

(15) (ii) If $A = \{B, F\}$ where B is a SO, F is a syntactic feature and $\text{Triggers}(B) \neq \varnothing$, then $\text{Triggers}(A) = \text{Triggers}(B) \setminus \{\text{uF}\}$ for some uninterpretable syntactic feature $\text{uF} \in \text{Triggers}(B)$.

There are two cases of Merge we must consider: $\text{Merge}(A, B)$ where A, B are both SOs, and $\text{Merge}(A, F)$ where A is an SO and F is a syntactic feature. The first case is the old Merge, which we call Merge_{SO} from now on. The second case is Merge_{F}, which we define as follows:

[8] An alternative we do not explore in this paper is to allow syntactic features themselves be SOs.

(11) a. Build TP.
PRES has SYN = { [T], [=*v*], [EPP], [uπ:_], [u#:_], [iC:nom] }.
man has SYN = { [N], [iπ:3], [i#:sg], [uC:_] }.
$W_i = \{\{$ PRES, { *v*, { falls, { the, man }}}}}

b. Agree applies, with PRES as the probe and *man* as the goal. They are replaced with PRES′ and *man*′. Since PRES has EPP, the DP *the man*′ is pied-piped to [Spec; TP].
PRES′ has SYN = { [T], [=*v*], [EPP], [uπ:3], [u#:sg], [iC:nom] }.
man′ has SYN = { [N], [iπ:3], [i#:sg], [uC:nom] }.
$W_j = \{$ { the, man′ }, { PRES′, { *v*, { falls, { the, man′ }}}}}

(12) a. Select PRES and *man*.
PRES has SYN = { [T], [=*v*], [EPP], [uπ:3], [u#:sg], [iC:nom] }.
man has SYN = { [N], [iπ:3], [i#:sg], [uC:nom] }.
$W_i = \{$PRES, man$\}$

b. Merge *man* with [iC:nom] from PRES.
$W_j = \{$PRES, { man, [iC:nom] }$\}$

c. Build TP, up to and including movement of *the man* to [Spec; TP]. Call the result TP_k.
$W_k = \{\underbrace{\{ \{ \text{the}, \{ \text{man}, [\text{iC:nom}] \} \}, \{ \text{PRES}, \{ v, \{ \text{falls}, \{ \text{the}, \{ \text{man}, [\text{iC:nom}] \} \}\}\}\}\}}_{\text{TP}_k}\}$

d. Merge TP_k with [iπ:3], then with [i#:sg], both from *man*.
$W_\ell = \{\{$ [i#:sg], { [iπ:3], TP_k }}$\}$

(16) Given any syntactic object X and syntactic feature F, where $\text{Triggers}(X) \neq \varnothing$, $\text{Merge}_\text{F}(X, F) = \{X, F\}$.

Finally, we modify Clause (iii) from the definition of derivations. The derive-by-Merge condition must be split in two cases: derive-by-Merge_SO, which is the old derive-by-Merge, and derive-by-Merge_F, which handles derivation by $\text{Merge}_\text{F}(A, F)$ for some SO A and syntactic feature F. Derive-by-Merge_F requires F to be part of some LIT contained in the workspace, but not necessarily contained in A. In other words, sideward Merge is allowed only for Merge_F.

(17) (derive-by-Merge_F) $\text{LA}_i = \text{LA}_{i+1}$ and the following conditions hold for some A, F:
a. $A \in W_i$,
b. There exists some lexical item token $X \in^+ W_i$ such that $X = \langle\langle \text{SEM}, \text{SYN}, \text{PHON}\rangle, k\rangle$ where $F \in \text{SYN}$, and
c. $W_{i+1} = (W_i \backslash \{A\}) \cup \{\text{Merge}_\text{F}(A, F)\}$.

We illustrate our implementation with (12), a derivation for the sentence *The man falls*. This derivation is based on Derivation (27) in Collins 2017, where the T head PRES Merges with the ϕ-features from *man* to form the complex T { PRES, [iϕ] } before Merging with *v*P. This is problematic, as $\text{Transfer}_\text{PF}$ cannot linearize the TP { { PRES, [iϕ] }, *v*P} because *v*P is neither a complement, as the complex T is not a LIT; nor is *v*P a specifier, as the complex T is not a set of SOs either. In our derivation (12), we let PRES Merge with its *v*P complement before Merging with the ϕ-features, avoiding the $\text{Transfer}_\text{PF}$ problem. The full derivation is in Appendix D.

6 Future work

In Section 4, we showed how extensions of `cands` can be evaluated against a common set of derivations, offering a quantitative comparison of their empirical coverages. Our evaluation setup can be scaled up quite easily, by curating a large-scale test set of derivations, which can then be used to evaluate `cands`-based implementations of many different theories. This kind of evaluation is familiar in the parsing literature, where parsers are evaluated on large datasets of syntactically annotated sentences known as *treebanks*, such as the Penn Treebank (Marcus et al., 1993), CCGbank (Hockenmaier and Steedman, 2002), the Redwoods treebank (Flickinger, 2011; Oepen et al., 2002, 2004), MGbank (Torr, 2017, 2018), among others.

While `cands` can check if C&S generates a given derivation, it cannot check if C&S generates some derivation that linearizes to a given PF. Obviously, syntacticians are equally if not more interested in problems of the latter type. For example, one might wish to check if a theory overgenerates, i.e., if it derives an ungrammatical sentence, or if it derives a grammatical sentence with an undesirable derivation. Solving this type of problems requires us to develop an algorithm that automatically explores the predictions from C&S, which is essentially a parser. There is a recent line of work on neural transition-based parsers, i.e. neural classifiers that take parser states as input and output parser transitions as output (Dyer et al., 2016; Yoshida and Oseki, 2022; Sartran et al., 2022). While these parsers are typically implemented with state-of-the-art neural architectures, they usually only support parsing for primitive grammars, such as PCFGs. As such, we hope to explore if neural transition-based parsers can be developed for more complex grammars, such as Minimalist Grammars (Stabler, 1997) and C&S. An even more challenging task is to develop methods to (semi)automatically derive a parser for an arbitrary extension of C&S.

Finally, `cands` brings us closer to the quantitative evaluation of the parsimony of C&S and relevant theories. For example, any `cands`-based implementation of some theory provides an upper bound for the *minimum description length (MDL)* of that theory. MDL can in turn be used to define a prior distribution over theories in a probabilistic setup (Berwick, 2015).

7 Conclusion

We present CANDS, a Rust implementation of Collins and Stabler's (2016; C&S) formalization of a minimalist syntactic theory. The core of CANDS is `cands`, a library. `cands` by itself can be used to explore predictions from the C&S system, and it can also be extended to implement other theoretical notions. We also present `derivck` and `derivexp`, two wrapper programs that allows the user to check and explore derivations with `cands` without having to program in Rust.

Computational implementation of syntactic theories greatly facilitates the evaluation of their empirical coverages, and forces the programmer to attend to the details and edge cases of the theories, which can be easily miscommunicated in textual descriptions of minimalist syntactic theory. In this paper, we show how CANDS can be integrated into a minimalist syntactician's typical workflow. We hope our work will benefit the minimalist syntax community, and we welcome suggestions and contributions, as our work is still under much development.

Acknowledgments

We would like to thank Yushi Sugimoto and other members of OsekiLab at the University of Tokyo for their helpful comments and suggestions at various stages of this project. We also want to thank Kyle Johnson, Faruk Akkuş, and three anonymous reviewers for their thoughtful feedback. All remaining errors are ours. This work was supported by JST PRESTO Grant Number JPMJPR21C2.

References

Adrian Akmajian. 1975. More evidence for an np cycle. *Linguistic Inquiry*, 6(1):115–129.

Susana Béjar and Milan Rezac. 2009. Cyclic agree. *Linguistic Inquiry*, 40(1):35–73.

Emily M Bender, Scott Drellishak, Antske Fokkens, Michael Wayne Goodman, Daniel P Mills, Laurie Poulson, and Safiyyah Saleem. 2010. Grammar prototyping and testing with the lingo grammar matrix customization system. In *Proceedings of the ACL 2010 system demonstrations*, pages 1–6.

Emily M Bender, Dan Flickinger, and Stephan Oepen. 2002. The grammar matrix: An open-source starter-kit for the rapid development of cross-linguistically consistent broad-coverage precision grammars. In *COLING-02: Grammar Engineering and Evaluation*.

Emily M Bender, Dan Flickinger, and Stephan Oepen. 2008. Grammar engineering for linguistic hypothesis testing. In *Proceedings of the Texas Linguistics Society X Conference: Computational linguistics for less-studied languages*, pages 16–36.

Robert C. Berwick. 2015. Mind the gap. In Ángel J Gallego and Dennis Ott, editors, *50 years later: Reflections on Chomsky's Aspects*. MIT Working Papers in Linguistics.

Manfred Bierwisch. 1963. *Grammatik des deutschen Verbs*. Akademie Verlag, Berlin.

Miriam Butt. 1999. A grammar writer's cookbook. *CSLI Lecture Notes*.

Noam Chomsky. 1995. *The Minimalist Program*. MIT press.

Noam Chomsky. 2001. Derivation by Phase. In *Ken Hale: A Life in Language*. The MIT Press.

Noam Chomsky. 2007. Approaching ug from below. *Interfaces+ recursion= language*, 89:1–30.

Chris Collins. 2017. Merge (x, y)={X, Y}. In Leah Bauke and Andreas Blümel, editors, *Labels and roots*, pages 47–68. De Gruyter Mouton.

Chris Collins and Edward Stabler. 2016. A formalization of minimalist syntax. *Syntax*, 19(1):43–78.

Amy Rose Deal. 2015. Interaction and satisfaction in φ-agreement. In *Proceedings of NELS*, volume 45, pages 179–192.

Chris Dyer, Adhiguna Kuncoro, Miguel Ballesteros, and Noah A. Smith. 2016. Recurrent neural network grammars. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 199–209, San Diego, California. Association for Computational Linguistics.

Dan Flickinger. 2011. Accuracy vs. robustness in grammar engineering. *Language from a cognitive perspective: Grammar, usage, and processing*, 201:31–50.

Antske Sibelle Fokkens. 2014. Enhancing empirical research for linguistically motivated precision grammars. *Saarbrücken: Saarland University*.

Sandiway Fong and Jason Ginsburg. 2019. 16Towards a Minimalist Machine. In *Minimalist Parsing*. Oxford University Press.

John Frampton. 2004. Copies, traces, occurrences, and all that. *Ms., Northeastern University*.

Jacqueline Guéron. 1980. On the syntax and semantics of pp extraposition. *Linguistic inquiry*, 11(4):637–678.

Julia Hockenmaier and Mark Steedman. 2002. Acquiring compact lexicalized grammars from a cleaner treebank. In *LREC*, volume 42, page 58.

Marcus Kracht. 1999. Adjunction structures and syntactic domains. In Hans-Peter Kolb and Uwe Mönnich, editors, *The Mathematics of Syntactic Structure: Trees and their Logics*. De Gruyter Mouton, Berlin, Boston.

Marcus Kracht. 2001. Syntax in chains. *Linguistics and Philosophy*, 24:467–530.

Marcus Kracht. 2008. On the logic of lgb type structures. part i: Multidominance structures. *Logics for linguistic structures*, pages 105–142.

Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. 1993. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330.

Stefan Müller. 2015. The coregram project: theoretical linguistics, theory development, and verification. *Journal of Language Modelling*, 3(1):21–86.

Stefan Müller. 1999. *Deutsche Syntax deklarativ*. Max Niemeyer Verlag, Berlin, Boston.

Stephan Oepen, Emily M Bender, Uli Callmeier, Dan Flickinger, and Melanie Siegel. 2002. Parallel distributed grammar engineering for practical applications. In *COLING-02: Grammar Engineering and Evaluation*.

Stephan Oepen, Dan Flickinger, Kristina Toutanova, and Christopher D Manning. 2004. Lingo redwoods: A rich and dynamic treebank for hpsg. *Research on Language and Computation*, 2:575–596.

David Pesetsky and Esther Torrego. 2007. The syntax of valuation and the interpretability of features. *Phrasal and clausal architecture*, pages 262–294.

Omer Preminger. 2014. *Agreement and Its Failures*. MIT Press.

Laurent Sartran, Samuel Barrett, Adhiguna Kuncoro, Miloš Stanojević, Phil Blunsom, and Chris Dyer. 2022. Transformer grammars: Augmenting transformer language models with syntactic inductive biases at scale. *Transactions of the Association for Computational Linguistics*, 10:1423–1439.

Edward Stabler. 1997. Derivational minimalism. In *Logical Aspects of Computational Linguistics: First International Conference, LACL'96 Nancy, France, September 23–25, 1996 Selected Papers 1*, pages 68–95. Springer.

John Torr. 2017. Autobank: a semi-automatic annotation tool for developing deep minimalist grammar treebanks. In *Proceedings of the Software Demonstrations of the 15th Conference of the European Chapter of the Association for Computational Linguistics*, pages 81–86.

John Torr. 2018. Constraining mgbank: Agreement, l-selection and supertagging in minimalist grammars. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 590–600.

Mettina Jolanda Arnoldina Veenstra. 1998. *Formalizing the minimalist program*. Ph.D. thesis, Rijksuniversiteit Groningen.

Kenneth Wexler and Peter W Culicover. 1980. *Formal principles of language acquisition*, volume 76. MIT press Cambridge, MA.

Ryo Yoshida and Yohei Oseki. 2022. Composition, attention, or both? In *Findings of the Association for Computational Linguistics: EMNLP 2022*, pages 5822–5834, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Olga Zamaraeva. 2021. *Assembling syntax: Modeling constituent questions in a grammar engineering framework*. University of Washington.

Olga Zamaraeva, Chris Curtis, Guy Emerson, Antske Fokkens, Michael Wayne Goodman, Kristen Howell, TJ Trimble, and Emily M Bender. 2022. 20 years of the grammar matrix: cross-linguistic hypothesis testing of increasingly complex interactions. *Journal of Language Modelling*, 10(1):49–137.

Hedde Zeijlstra. 2012. There is only one way to agree. *The linguistic review*, 29(3):491–539.

Arnold M Zwicky, Joyce Friedman, Barbara C Hall, and Donald E Walker. 1965. The mitre syntactic analysis procedure for transformational grammars. In *Proceedings of the November 30–December 1, 1965, fall joint computer conference, part I*, pages 317–326.

A List of C&S definitions that are implemented in `cands`

Table 2 contains a list of all definitions in C&S. For each definition, we indicate whether it is implemented in `cands`.

We left four groups of definitions from C&S unimplemented. The first group consists of tentative definitions; they are presented in earlier parts of the C&S paper, and eventually replaced by more complete definitions later in the paper. Specifically, this group consists of Definitions 8 (SO) and 14 (derivation), which are replaced by Definitions 37 and 38. We implement the latter definitions instead of the former ones.

The second group of unimplemented definitions simply cannot be implemented. This applies to Definitions 15, 15′ and 23. These define the concept of the derivability of a given SO or workspace. Derivability itself is a binary value, either true or false – it is a trivial definition that does not need an implementation. Presumably, it is more interesting to implement a function that would compute the derivability from a given SO or workspace. To implement such a function, we need to create a parser for the C&S system. This is beyond the scope of our paper.

The third group of unimplemented definitions are unnecessary to implement. This applies to Definition 25, which defines trigger features. Trigger features are just a special name to designate a certain group of features for a particular $\mathrm{Triggers}$ implementation. As the concept is purely expository, it has no place in our implementation of C&S.

The last group of unimplemented definitions concern occurrences (Definitions 16, 17, 18, 20, 22) and chain-based SOs (Definitions 16′, 7′, 13′, 14′, 15′), which are only partially explored in C&S as a digression from their full formalization of a theory of token-based SOs. We leave their implementations to future work.

No.	Definition	In `cands`?
Section 2: Preliminary definitions		
1	Universal Grammar	Yes
2	Lexical item	Yes
3	Lexicon	Yes
4	I-language	Yes
5	Lexical item token	Yes
6	Lexical array	Yes
7	Syntactic object (old)	No
8	Immediate containment (SO)	Yes
9	Containment	Yes
Section 3: Workspaces, Select, and Merge		
10	Stage	Yes
	Workspace	Yes
11	Roothood	Yes
12	Select	Yes
13	Merge	Yes
14	Derivation (old)	No
15	Derivability from lexicon	No
Section 4: Occurrences		
16	Position	No
17	Occurrence	No
18	Immediate containment (occurrence)	No
19	Sisterhood (SO)	Yes
20	Sisterhood (occurrence)	No
21	C-command (SO)	Yes
	Asymmetric c-command (SO)	Yes
22	C-command (occurrence)	No
Section 5: Digression		
$16'$	Path (chain-based)	No
$7'$	SO (chain-based)	No
$13'$	Merge (chain-based)	No
$14'$	Derivation (chain-based)	No
$15'$	Derivability from lexicon (chain-based)	No

No.	Definition	In `cands`?
Section 6: General Properties of Derivations		
23	Derivability	No
24	Binary branching	Yes
Section 7: Labels		
25	Trigger feature	No
26	Triggers	Yes
27	Triggered Merge	Yes
28	Label	Yes
29	Maximal projection	Yes
30	Minimal projection	Yes
31	Intermediate projection	Yes
32	Complement	Yes
33	Specifier	Yes
Section 8: Transfer		
34	Transfer	Yes
35	Strong phasehood	Yes
36	Cyclic-Transfer	Yes
37	Syntactic object (new)	Yes
38	Derivation (new)	Yes
Section 9: $\text{Transfer}_{\text{LF}}$		
39	$\text{Transfer}_{\text{LF}}$	Yes
Section 10: $\text{Transfer}_{\text{PF}}$		
40	Finality	Yes
41	$\text{Transfer}_{\text{PF}}$	Yes
Section 13: Convergence		
42	Convergence and crash at the CI interface	Yes
43	Convergence and crash at the SM interface	Yes
44	Convergence and crash	Yes

Table 2: List of definitions in C&S. For each definition, we indicate whether it is implemented in `cands`.

B Implementing the extensions of `cands` for PP extraposition

In Section 4, we described two extensions of C&S, where each extension is created by adding one constraint into the C&S system. In this section, we describe our implementation of these extensions in more detail.

B.1 Theory 1

Theory 1 is the extension of C&S by the Derivational Subject Condition (DSC), defined in (5). DSC further constricts the derive-by-Merge condition, specifically the internal Merge case.

The derive-by-Merge condition is checked by the `derive_by_merge` function, which is used by the `is_derivation` to check if each non-initial stage is derived from its previous stage by an appropriate application of a syntactic operation, including Merge. We implement DSC inside the `derive_by_merge` function. The pseudocode for `derive_by_merge` as well as the DSC is provided in Algorithm 1. The for-loop starting on line 6 checks for internal Merge, and the for-loop starting on line 13 checks for external Merge. Once an appropriate pair of SOs A, B is found in either for-loop, the function returns true from within that loop. The DSC is thus implemented in the for-loop for internal Merge. At line 9, we check the negation of DSC; if the DSC is violated, the if-statement is executed, and the current iteration of the for-loop will be skipped (also known as a *continue*-statement). As such, the return-statement on line 12 is unreachable in the current iteration. This implements the DSC.

B.2 Theory 2

Theory 2 is the extension of C&S by the Representational Subject Condition (RSC), defined in (6). RSC is checked for every stage in the derivation.

We implement RSC is the `is_derivation` function, whose pseudocode is provided in Algorithm 2. The for-loop starting on line 7 checks whether each pair of consecutive stages is derived-by-Select, Merge or Transfer. The if-statement on line 8 checks if neither of these three syntactic operations derive the second stage from the first, in which case the function returns false. RSC further constraints this check. If RSC is violated, the if-statement on line 15 will execute, and the function returns false.

Input: Two stages $S_1 = \langle \mathrm{LA}_1, W_1 \rangle$ and $S_2 = \langle \mathrm{LA}_2, W_2 \rangle$
Output: true iff S_2 is derived-by-Merge from S_1

```
1  if LA_1 ≠ LA_2 then
2      return false;
3  if W_1 is empty then
4      return false;
5  foreach A ∈* W_1 do
6      foreach B such that B ∈* A do
           /* ===== DSC begins =====                                   */
7          Calculate |occ_A(B)| ;
8          Calculate Σ_X |occ_A(X)|, the sum of |occ_A(X)| for all [Spec; TP] X ∈* A such that
             B ∈* X ;
9          if |occ_A(B)| ≤ Σ_X |occ_A(X)| then
10             Skip to the next pair of A, B;
           /* ====== DSC ends ======                                   */
11         if W_2 = W_1 \ {A, B} ∪ {Merge(A, B)} then
12             return true;
13     foreach B such that B ∈ W_1 do
14         if W_2 = W_1 \ {A, B} ∪ {Merge(A, B)} then
15             return true;
16 return false;
```

Algorithm 1: Pseudocode for the `derive_by_merge` function. The implementation of DSC is between lines 7–10, inclusive on both ends.

Input: an I-language $\mathrm{IL} = \langle \mathrm{Lex}, \mathrm{UG} \rangle$, and a sequence of stages $S = \langle S_1, \cdots, S_n \rangle$, with $S_i = \langle \mathrm{LA}_i, W_i \rangle$ for each $i \in [n]$

Output: true iff S is a derivation from IL

```
1  if S is empty then
2      return false;
3  if there is some LIT X ∈ LA_1 that is not contained in Lex then
4      return false;
5  if W_1 is not empty then
6      return false;
7  foreach i < n do
8      if S_{i+1} is not derived-by-Select from S_i and S_{i+1} is not derived-by-Merge from S_i and S_{i+1} is
         not derived-by-Transfer from S_i then
9          return false;
       /* ===== RSC begins =====                                          */
10     foreach R ∈ W_{i+1} do
11         let S = the set of all S ∈* R such that S is [Spec; TP];
12         let X = the set of all X ∈* S for some S ∈ S;
13         Calculate |occ_R(S)| for each S ∈ S ;
14         Calculate |occ_R(X)| for each X ∈ X ;
15         if |occ_R(X)| ≠ |occ_R(S)| for any S ∈ S and any X ∈ X then
16             return false;
       /* ====== RSC ends ======                                          */
17 return true;
```

Algorithm 2: Pseudocode for the `is_derivation` function. The implementation of RSC is between lines 10–16, inclusive on both ends.

C Full derivations for the extraposition sentences

Here, we provide the full derivations for the sentences (1a), (1b) and (1c) in Section 4. These derivations were sketched in the main paper as (2), (3) and (4).

We assume the lexicon in Table 3. The semantic, syntactic and phonological features of our UG are the unions of the semantic, syntactic and phonological features over the LIs in our lexicon. The syntactic features include (a) category features of the form [α], where α is a syntactic category; (b) selectional features of the form [=α], where α is a syntactic category; (c) EPP-feature [EPP], and (d) *wh*-features [u*wh*] and [i*wh*]. Selectional features, EPP-feature and [u*wh*] are trigger features. A selectional feature [=α] can be checked by Merging with some SO whose label bears the category feature [α]. An EPP-feature can be checked by Merging with some SO. [u*wh*] can be checked by Merging with some SO whose labels bears [i*wh*]. We use two pairs of heads X and Y to handle extraposition; we use $\mathrm{X_{T,P}}$ and $\mathrm{Y_T}$ to handle PP extraposition from TP and use $\mathrm{X_{V,P}}$ and $\mathrm{Y_V}$ to handle PP extraposition from VP.

The derivations (18), (19) and (20) are for the sentences (1a), (1b) and (1c) respectively. For each stage S_i, we describe the syntactic operation by which S_i is derived, and we show its workspace W_i. We omit Select for brevity. Transferred SOs are struck out.

(18)

a. $\mathrm{Merge}(\textit{bothered}, \textit{me})$.
$W_1 = \{\underbrace{\{\text{ bothered, me }\}}_{\mathrm{VP}}\}$.

b. $\mathrm{Merge}(v^*, \mathrm{VP})$.
$W_2 = \{\underbrace{\{\ v^*, \mathrm{VP}\ \}}_{v^*\mathrm{P}_1}\}$.

c. $\mathrm{Transfer}(v^*\mathrm{P}_1, \mathrm{VP})$.
$W_3 = \{\underbrace{\{\ v^*, \cancel{\mathrm{VP}}\ \}}_{v^*\mathrm{P}_2}\}$.

d. $\mathrm{Merge}(\textit{about}, \textit{Mary})$.
$W_4 = \{\underbrace{\{\text{ about, Mary }\}}_{\mathrm{PP}}, v^*\mathrm{P}_2\}$.

e. $\mathrm{Merge}(\textit{story}, \mathrm{PP})$.
$W_5 = \{\underbrace{\{\text{ story, PP }\}}_{\mathrm{NP}}, v^*\mathrm{P}_2\}$.

f. $\mathrm{Merge}(a, \mathrm{NP})$.
$W_6 = \{\underbrace{\{\text{ a, NP }\}}_{\mathrm{DP}}, v^*\mathrm{P}_2\}$.

g. $\mathrm{Merge}(v^*\mathrm{P}_2, \mathrm{DP})$.
$W_7 = \{\underbrace{\{\text{ DP, }v^*\mathrm{P}_2\ \}}_{v^*\mathrm{P}_3}\}$.

h. $\mathrm{Merge}(\mathrm{PAST}_{v^*}, v^*\mathrm{P}_3)$.
$W_8 = \{\underbrace{\{\ \mathrm{PAST}_{v^*}, v^*\mathrm{P}_3\ \}}_{\mathrm{TP}_1}\}$.

i. $\mathrm{Merge}(\mathrm{TP}_1, \mathrm{DP})$.
$W_9 = \{\underbrace{\{\text{ DP, TP}_1\ \}}_{\mathrm{TP}_2}\}$.

j. $\mathrm{Merge}(\mathrm{X_{T,P}}, \mathrm{TP}_2)$.
$W_{10} = \{\underbrace{\{\text{ X}_{\mathrm{T,P}}\text{, TP}_2\ \}}_{\mathrm{XP}_1}\}$.

k. $\mathrm{Merge}(\mathrm{XP}_1, \mathrm{PP})$.
$W_{11} = \{\underbrace{\{\text{ PP, XP}_1\ \}}_{\mathrm{XP}_2}\}$.

l. $\mathrm{Merge}(\mathrm{Y_T}, \mathrm{XP}_2)$.
$W_{12} = \{\underbrace{\{\text{ Y}_{\mathrm{T}}\text{, XP}_2\ \}}_{\mathrm{YP}_1}\}$.

m. $\mathrm{Merge}(\mathrm{YP}_1, \mathrm{TP}_2)$.
$W_{13} = \{\underbrace{\{\text{ TP}_2\text{, YP}_1\ \}}_{\mathrm{YP}_2}\}$.

n. $\mathrm{Merge}(\mathrm{C}, \mathrm{YP}_2)$.
$W_{14} = \{\underbrace{\{\text{ C, YP}_2\ \}}_{\mathrm{CP}}\}$.

o. $\mathrm{Transfer}(\mathrm{CP}, \mathrm{CP})$.
$W_{15} = \{\cancel{\mathrm{CP}}\}$.

(19) a. $\text{Merge}(\textit{about}, \textit{Mary})$.
$W_1 = \{\underbrace{\{\text{ about, Mary }\}}_{\text{PP}}\}$.

b. $\text{Merge}(\textit{story}, \text{PP})$.
$W_2 = \{\underbrace{\{\text{ story, PP }\}}_{\text{NP}}\}$.

c. $\text{Merge}(\textit{a}, \text{NP})$.
$W_3 = \{\underbrace{\{\text{ a, NP }\}}_{\text{DP}}\}$.

d. $\text{Merge}(\textit{appeared}, \text{DP})$.
$W_4 = \{\underbrace{\{\text{ appeared, DP }\}}_{\text{VP}}\}$.

e. $\text{Merge}(\text{X}_{\text{V,P}}, \text{VP})$.
$W_5 = \{\underbrace{\{\text{ X}_{\text{V,P}}\text{, VP }\}}_{\text{XP}_1}\}$.

f. $\text{Merge}(\text{XP}_1, \text{PP})$.
$W_6 = \{\underbrace{\{\text{ PP, XP}_1\text{ }\}}_{\text{XP}_2}\}$.

g. $\text{Merge}(\text{Y}_{\text{V}}, \text{XP}_2)$.
$W_7 = \{\underbrace{\{\text{ Y}_{\text{V}}\text{, XP}_2\text{ }\}}_{\text{YP}_1}\}$.

h. $\text{Merge}(\text{YP}_1, \text{VP})$.
$W_8 = \{\underbrace{\{\text{ VP, YP}_1\text{ }\}}_{\text{YP}_2}\}$.

i. $\text{Merge}(v, \text{YP}_2)$.
$W_9 = \{\underbrace{\{\text{ }v\text{, YP}_2\text{ }\}}_{v\text{P}}\}$.

j. $\text{Merge}(\text{PAST}_v, v\text{P})$.
$W_{10} = \{\underbrace{\{\text{ PAST}_v\text{, }v\text{P }\}}_{\text{TP}_1}\}$.

k. $\text{Merge}(\text{TP}_1, \text{DP})$.
$W_{11} = \{\underbrace{\{\text{ DP, TP}_1\text{ }\}}_{\text{TP}_2}\}$.

l. $\text{Merge}(\text{C}, \text{TP}_2)$.
$W_{12} = \{\underbrace{\{\text{ C, TP}_2\text{ }\}}_{\text{CP}}\}$.

m. $\text{Transfer}(\text{CP}, \text{CP})$.
$W_{13} = \{\cancel{\text{CP}}\}$.

(20) a. Same as (19) up to and including (19m), but replace *Mary* with *who*.
$W_{13} = \{\cancel{\text{CP}_1}\}$.

b. $\text{Merge}(\textit{know}, \text{CP}_1)$.
$W_{14} = \{\underbrace{\{\text{ know, CP}_1\text{ }\}}_{\text{VP}}\}$.

c. $\text{Merge}(v^*, \text{VP})$.
$W_{15} = \{\underbrace{\{\text{ }v^*\text{, VP }\}}_{v^*\text{P}_1}\}$.

d. $\text{Transfer}(v^*\text{P}_1, \text{VP})$.
$W_{16} = \{\underbrace{\{\text{ }v^*\text{, }\cancel{\text{VP}}\text{ }\}}_{v^*\text{P}_2}\}$.

e. $\text{Merge}(v^*\text{P}_2, \textit{we})$.
$W_{17} = \{\underbrace{\{\text{ we, }v^*\text{P}_2\text{ }\}}_{v^*\text{P}_3}\}$.

f. $\text{Merge}(\text{PRES}_{v^*}, v^*\text{P}_3)$.
$W_{18} = \{\underbrace{\{\text{ PRES}_{v^*}\text{, }v^*\text{P}_3\text{ }\}}_{\text{TP}_1}\}$.

g. $\text{Merge}(\text{TP}_1, \text{DP})$.
$W_{19} = \{\underbrace{\{\text{ DP, TP}_1\text{ }\}}_{\text{TP}_2}\}$.

h. $\text{Merge}(\text{C}, \text{TP}_2)$.
$W_{20} = \{\underbrace{\{\text{ C, TP}_2\text{ }\}}_{\text{CP}_2}\}$.

i. $\text{Transfer}(\text{CP}_2, \text{CP}_2)$.
$W_{21} = \{\cancel{\text{CP}_2}\}$.

LI	**SEM**	**SYN**	**PHON**
Mary	{[Mary]}	{[D]}	⟨[Mary]⟩
me	{[me]}	{[D]}	⟨[me]⟩
we	{[we]}	{[D]}	⟨[we]⟩
who	{[who]}	{[D], [i*wh*]}	⟨[who]⟩
about	{[about]}	{[P], [=D]}	⟨[about]⟩
story	{[story]}	{[N], [=P]}	⟨[story]⟩
a	{[a]}	{[D], [=N]}	⟨[a]⟩
bothered	{[bothered]}	{[V], [=D]}	⟨[bothered]⟩
appeared	{[appeared]}	{[V], [=D]}	⟨[appeared]⟩
know	{[know]}	{[V], [=C]}	⟨[know]⟩
*v**	{[*v**]}	{[*v**], [=V], [=D]}	⟨⟩
v	{[*v*]}	{[*v*], [=V]}	⟨⟩
$X_{T,P}$	{[X]}	{[X], [=T], [=P]}	⟨⟩
$X_{V,P}$	{[X]}	{[X], [=V], [=P]}	⟨⟩
Y_T	{[Y]}	{[T], [=X], [=T]}	⟨⟩
Y_V	{[Y]}	{[V], [=X], [=V]}	⟨⟩
PRES_{v*}	{[PRES]}	{[T], [=*v**], [EPP]}	⟨⟩
PAST_{v*}	{[PAST]}	{[T], [=*v**], [EPP]}	⟨⟩
PAST_{v}	{[PAST]}	{[T], [=*v*], [EPP]}	⟨⟩
C	{[C]}	{[C], [=T]}	⟨⟩
Q	{[Q]}	{[C], [=T], [u*wh*]}	⟨⟩

Table 3: Lexicon for the derivations (18), (19) and (20). For example, the LI *Mary* is a 3-tuple $\langle \text{SEM}, \text{SYN}, \text{PHON} \rangle$ where $\text{SEM} = \{[\text{Mary}]\}$, $\text{SYN} = \{[\text{D}]\}$ and $\text{PHON} = \langle [\text{Mary}] \rangle$.

D Full derivations for *The man falls*

Here, we provide the full derivations for the sentence *The man falls* in Section 5. These derivations were sketched in the main paper as (11) and (12).

We assume the lexicon in Table 4. Again, the semantic, syntactic and phonological features of our UG are the unions of the semantic, syntactic and phonological features over the LIs in our lexicon.

The derivations (21) and (22) are for the sentence *The man falls* in our implementations of Chomsky (2001) and Collins (2017) respectively. We omit most applications of Select for brevity, except at the beginning of (22). There, it is important that the tense head PRES be selected near the beginning of the derivation, before any Merge takes place.

LI	SEM	SYN	PHON
the	{[the]}	{[D], [=N]}	⟨[the]⟩
falls	{[falls]}	{[V], [=D]}	⟨[falls]⟩
v	{[*v*]}	{[*v*], [=V]}	⟨⟩
C	{[C]}	{[C], [=T]}	⟨⟩
Unique to our Chomsky 2001 implementation:			
man	{[man]}	{[N], [iπ:3], [i#:sg], [uC:_]}	⟨[man]⟩
man′	{[man]}	{[N], [iπ:3], [i#:sg], [uC:nom]}	⟨[man]⟩
PRES	{[PRES]}	{[T], [=*v*], [EPP], [uπ:_], [u#:_], [iC:nom]}	⟨⟩
PRES′	{[PRES]}	{[T], [=*v*], [EPP], [uπ:3], [u#:sg], [iC:nom]}	⟨⟩
Unique to our Collins 2017 implementation:			
man	{[man]}	{[N], [iπ:3], [i#:sg], [uC:nom]}	⟨[man]⟩
PRES	{[PRES]}	{[T], [=*v*], [EPP], [uπ:3], [u#:sg], [iC:nom]}	⟨⟩

Table 4: Lexicon for the derivations (21) and (22).

(21) a. Merge(*the*, *man*).
$W_1 = \{\underbrace{\{\text{ the, man }\}}_{\text{DP}}\}$.

b. Merge(*falls*, DP).
$W_2 = \{\underbrace{\{\text{ falls, }\{\text{ the, man }\}\ \}}_{\text{VP}}\}$.

c. Merge(v, VP).
$W_3 = \{\underbrace{\{\text{ }v\text{, }\{\text{ falls, }\{\text{ the, man }\}\ \}\ \}}_{v\text{P}}\}$.

d. Merge(PRES, vP).
$W_4 = \{\underbrace{\{\text{ PRES, }\{\text{ }v\text{, }\{\text{ falls, }\{\text{ the, man }\}\ \}\ \}\ \}}_{\text{TP}_1}\}$.

e. Agree(PRES, *man*).
$W_5 = \{\underbrace{\{\text{ }\{\text{ the, man}'\text{ }\}\text{, }\{\text{ PRES}'\text{, }\{\text{ }v\text{, }\{\text{ falls, }\{\text{ the, man}'\text{ }\}\ \}\ \}\ \}\ \}}_{\text{TP}_2}\}$.

f. Merge(C, TP_2).
$W_6 = \{\underbrace{\{\text{ C, }\{\text{ }\{\text{ the, man}'\text{ }\}\text{, }\{\text{ PRES}'\text{, }\{\text{ }v\text{, }\{\text{ falls, }\{\text{ the, man}'\text{ }\}\ \}\ \}\ \}\ \}\ \}}_{\text{CP}}\}$.

g. Transfer(CP, CP).
$W_7 = \{\cancel{\text{CP}}\}$.

(22) a. Select(*man*).
$W_1 = \{\text{man}\}$.

b. Select(PRES).
$W_2 = \{\text{man}, \text{PRES}\}$.

c. Merge(*man*, [iC:nom]). [iC:nom] is in the SYN of PRES.
$W_3 = \{\underbrace{\{\text{ man, [iC:nom] }\}}_{\text{N}}, \text{PRES}\}$.

d. Merge(*the*, N).
$W_4 = \{\underbrace{\{\text{ the, \{ man, [iC:nom] } \}}_{\text{DP}}, \text{PRES}\}$.

e. Merge(*falls*, DP).
$W_5 = \{\underbrace{\{\text{ falls, \{ the, \{ man, [iC:nom] } \} }\}}_{\text{VP}}, \text{PRES}\}$.

f. Merge(v, VP).
$W_6 = \{\underbrace{\{\text{ } v\text{, \{ falls, \{ the, \{ man, [iC:nom] } \} \} }\}}_{v\text{P}}, \text{PRES}\}$.

g. Merge(PRES, vP).
$W_7 = \{\underbrace{\{\text{ PRES, \{ } v\text{, \{ falls, \{ the, \{ man, [iC:nom] } \} \} \} }\}}_{\text{TP}_1}\}$.

h. Merge(TP_1, DP).
$W_8 = \{\underbrace{\{\text{ \{ the, \{ man, [iC:nom] } \}, \{ PRES, \{ } v\text{, \{ falls, \{ the, \{ man, [iC:nom] } \} \} \} \} \} }\}}_{\text{TP}_2}\}$.

i. Merge(TP_2, [iπ:3]). [iπ:3] is in the SYN of *man*.
$W_9 = \{\underbrace{\{\text{ [i}\pi\text{:3], \{ DP, \{ PRES, } v\text{P } \} \} }\}}_{\text{TP}_3}\}$.

j. Merge(TP_3, [i#:sg]). [i#:sg] is in the SYN of *man*.
$W_{10} = \{\underbrace{\{\text{ [i\#:sg], \{ [i}\pi\text{:3], \{ DP, \{ PRES, } v\text{P } \} \} \} }\}}_{\text{TP}_4}\}$.

k. Merge(C, TP_4).
$W_{11} = \{\underbrace{\{\text{ C, \{ [i\#:sg], \{ [i}\pi\text{:3], \{ DP, \{ PRES, } v\text{P } \} \} \} \} }\}}_{\text{CP}}\}$.

l. Transfer(CP, CP).
$W_{12} = \{\text{CP}\}$.